Program Comprehension as Fact Finding

Thomas D. LaToza David Garlan James D. Herbsleb Brad A. Myers
School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213
{tlatoza, garlan, jdh, bam}@cs.cmu.edu

ABSTRACT

Little is known about how developers think about design during code modification tasks or how experienced developers' design knowledge helps them work more effectively. We performed a lab study in which thirteen developers worked for 3 hours understanding the design of a 54 KLOC open source application. Participants had from 0 to 10.5 years of industry experience and were grouped into three "experts" and ten "novices." We observed that participants spent their time seeking, learning, critiquing, explaining, proposing, and implementing *facts* about the code such as "getFoldLevel has effects". These facts served numerous roles, such as suggesting changes, constraining changes, and predicting the amount of additional investigation necessary to make a change. Differences between experts and novices included that the experts explained the root cause of the design problem and made changes to address it, while novice changes addressed only the symptoms. Experts did not read more methods but also did not visit some methods novices wasted time understanding. Experts talked about code in terms of abstractions such as "caching" while novices more often described code statement by statement. Experts were able to implement a change faster than novices. Experts perceived problems novices did not and were able to explain facts novices could not. These findings have interesting implications for future tools.

Categories and Subject Descriptors

D.2.6 [Software Engineering]: Programming Environments; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement.

General Terms

Design, Human Factors.

Keywords

Science of design, program comprehension, code navigation, empirical study, expertise, reverse engineering.

1. INTRODUCTION

Studies of programming have long built program comprehension models to describe how developers locate features, test hypotheses, navigate through code, or mentally represent small snippets of code [6]. However, these studies have not looked at software engineering activities abstracted from development environment features or implementation details of programs. Software engineering teaches that developers work with design decisions describing possible alternatives and dependency relationships between them [15][17], and developers are told to apply information hiding to prevent likely anticipated changes from rippling through a system [15], to refactor code clones to allow a single decision to be changed in one place, to write modular specifications allowing reasoning in isolation of the rest of the system, and to respect architectural styles [18] to prevent architectural drift and erosion [16].

There is a growing interest in better understanding and testing claims about how software engineering tools and principles help developers [8]. One approach is to apply program comprehension models to describe the influence of tools and principles on how developers work. But despite a long history, program comprehension models have had little success in realizing this goal [6]. For example, although claims that coupling makes code harder to comprehend have been indirectly supported by version control studies, little is known about the mechanisms by which coupling causes developers to work differently. A better understanding grounded in a program comprehension model might lead to better metrics for measuring coupling and better tools for alleviating it. But the mismatch between software engineering's interest in design decisions and existing program comprehension models makes this challenging.

Program comprehension models might also more accurately describe how developers work by modeling how developers think about design. A long tradition of studies in cognitive science has established that experts perform better not because they are smarter but because they have knowledge which novices lack (e.g., [3]). Studies of programmers have also found these differences, but have mostly studied knowledge in the form of highly local code idioms such as for loops (e.g., [5]). Software engineering suggests that developers have a wide variety of knowledge about good design in the form of abstractions such as design patterns [6] and architectural styles [18]. But little is known about this knowledge or how it helps developers work more effectively. A better understanding might lead to better guidelines for training software engineers and tools to help developers who have not yet learned this knowledge.

We conducted a study to understand how developers perform challenging code modification tasks and the effects of experience on this process. In two lab tasks, we provided participants with criticisms of the current design of the jEdit open source text editor, and instructed them to improve the design. Since the prior research had not yet identified the key variables with any degree of confidence, our study was conducted in an exploratory, open-ended way. We observed in detail how different developers approached the tasks, which allowed us to observe patterns and identify key variables for future experimental studies. We addressed three research questions:

- How does experience affect changes made to code?

ESEC-FSE'07, September 3–7, 2007, Cavat near Dubrovnik, Croatia.

- How does experience affect how developers work?
- How do developers reason about design during coding tasks?

We found that:

- Experts' changes addressed the cause of the problems while novices' changes addressed the symptoms.
- Experts made better decisions about which methods were relevant, they talked about the code using abstractions rather than statement-by-statement descriptions, they explained facts novices were unable to explain, and they implemented a change more quickly than novices.
- Developers described the design using facts which took several forms and served a number of roles.

2. RELATED WORK

Previous research shows that developers seek information in code to generate and test *what*, *why*, and *how* hypotheses while rapidly switching between control flow, data flow, and domain model representations of the program [20]. Developers search for relevant focus points to investigate, relate information to these points by investigating neighboring statements or methods. They collect this information to make a change to the code [11], forming good or bad relevancy perceptions based on the quality of cues such as identifier names, comments, and documentation. Developers ask design questions about the purpose and intended behavior of code, whether a wrong value was anticipated and ignored or overlooked, and the consequences of design decisions [9].

Numerous studies have found that developers do not mentally represent source code literally but recognize instances of schemas (e.g., iterating over a collection). *Schemas* are templates with slots filled in with situation-specific information. For example, developers apply `for`-loop schemas and sometimes forget specific information, such as recalling `i` instead of `j` for a loop index variable [5].Studies of expertise in other domains have found that many of the advantages of experts arise from their large library of schemas. For example, while chess experts remember realistic boards better than novices, their advantages vanish for random boards [3]. This and other results suggest that experts "chunk" what they perceive to mentally represent it in memory as schema instantiations. While there is much evidence for the existence and importance of schemas, studies of schemas in programming have been limited to highly localized code idioms (e.g., `for` loops) and have not investigated schemas at the level of design, which we wanted to investigate in our study.

Several studies have found differences between experienced and inexperienced developers working with code. Experts debug faster by generating better hypotheses while studying less code [7]. Experts write down low level information while novices write down higher level information [4]. Experts better understand code before changing it and better choose when to instantiate schemas. Experts select from multiple strategies for accomplishing tasks, are capable of generating multiple alternatives before making a choice, and design top-down more from high level ideas to low level ideas for familiar and simple problems [6]. Our study augments these results by adding differences during design.

A comparison of successful and unsuccessful behavior on programming tasks found successful participants had more programming experience than unsuccessful participants [14]. Unsuccessful participants made changes in one place that should have been scattered. These results suggest developers do not notice information unless they are searching for it. Successful participants created more detailed plans of changes to make before implementing them and reinvestigated methods less frequently. Successful participants performed more keyword and cross-reference searches for information rather than browsing or scrolling based on guesses. However, the five participants in that study had only 1 to 5 years programming experience and limited (if any) industry experience. This suggests that the results describe differences only between very little and little experience unlike our study where participants were more varied in their level of experience.

All of this existing work has revealed interesting results about how developers navigate code and mentally represent small snippets of code, but little is known about how developers uncover design or propose design changes and how these processes are affected by experience, which is the purpose of our study.

3. METHOD

We conducted an exploratory lab study where participants worked on two tasks for 1.5 hours each. The tasks were challenging and involved changes to a real open source application. We recorded participants' activity using think-aloud, video, and Eclipse instrumentation to get a full picture of what participants were doing.

3.1 Study Design

We recruited developers with diverse levels of experience, and brought them into the lab to observe their work in detail. A lab study had several advantages over a field study. We could compare participants' behavior on exactly the same tasks, use tasks designed to require understanding design, and control for prior experience with the application. We controlled for ordering effects between tasks by assigning half of the participants to receive each task first and ensured that there were experienced and novice participants in both conditions. An exploratory, observational study, rather than a controlled experiment, let us build a model of developer activity and differences suggested by it that we did not know beforehand. Our quantitative comparisons between experts and novices are not a controlled experiment because we picked dependent variables post-hoc from qualitative analysis of participant activity. We chose 13 participants, rather than a larger number (which might have resulted in statistically significant differences), to make manually transcribing and analyzing the voluminous transcripts feasible.

We initially planned to investigate the effect of providing architectural information on how developers work with code. We provided half of our participants with a component and connector [2] diagram that we reverse engineered. While these participants read the diagram at the beginning of the task, most used the diagram only to generate and test hypotheses about how classes were connected or as scratch paper to draw callgraphs or write down method names. We were unable to observe any differences about how the developers were working that could be attributed to having the diagram. We thus do not consider these diagrams further.

3.2 Participants

Thirteen participants were recruited from undergraduate students, masters students, doctoral students, and staff at Carnegie Mellon University who reported that they (1) had at least two programming internships or fulltime development experience and (2) were

Table 1. Participants' self-reported experience with medians for novices and experts. We assume internships lasted 1/4 of a year. For the experience columns on the right, 1 is the most experience and 7 the least.

Participant		yrs industry experience	KLOC largest program edited	yrs Java experience	Design patterns	Architectural styles	Refactoring tools	Code navigation proficiency	Enjoy designing
NOVICES	a	0 (research)	10	4	3	7	6	2	6
	b	0 (research)	7.5	3	3	1	1	2	1
	c	0.5	1	few	3	4	3	4	1
	d	1.5	75	5	2	3	1	3	1
	e	2	2	1	2	3	1	1	1
	f	2.5	1	2	3	6	7	1	1
	g	2.5	10	8	2	4	4	1	2
	h	2.5	136	4	2	2	2	2	2
	i	3	2	4	4	6	1	1	1
	j	3	10	6	2	6	2	4	1
		2.25	**8.75**	**4**	**3**	**4**	**2**	**2**	**1**
EXPERTS	K	3	100	7	1	1	1	1	1
	L	10	100	10	1	1	1	2	1
	M	10.5	500	3	1	2	6	2	2
		10	**100**	**7**	**1**	**1**	**1**	**2**	**1**

comfortable programming in Java. Industry experience and self-reported expertise data were collected with a short demographic survey completed when potential participants responded to our recruiting materials. Participants also rated their experience with design patterns, architectural styles, and refactoring tools, their perceived proficiency navigating code, and the degree to which they enjoyed designing (Table 1). Participants were asked to give the size of the largest program they had worked on. The low responses of several participants to this question suggest that they may have had inaccurate knowledge or misunderstood the question. Two participants who responded to our recruiting materials had no industry experience. Both were graduate students who reported significant research programming experience, so we accepted them for our study. Ten participants reported they had used Eclipse before, one reported she had not, and two were not asked about their experience using Eclipse. This suggests our results do not reflect challenges learning Eclipse.

Participants included one undergraduate student, four masters students, seven doctoral students, and one staff member. Participants had industry experience on a wide spectrum of applications including databases, banking software, and operating systems. Twelve males and one female participated. Participants were paid for their time. Table 1 shows the self-reported experience sorted by years of industry experience. We refer to participants as "experts" and "novices" for brevity. However, although the "novices" had limited industry experience, they still had substantial programming experience and should not be confused with novice programmers. Two participants (L, M) were labeled experts because they had far more experience than the novices. We labeled a third participant (K) an expert because he had slightly more experience than the novices and made the same changes as the other experts on one of the tasks. We refer to novices by lowercase letters and experts by uppercase letters.

One expert (L) had participated in an earlier study using the same application we used. Any advantages this participant had are potentially attributable to greater knowledge about the application rather than experience. However, we believe the effect of this contamination is minimal since our task required an understanding of an entirely different part of the application than the previous study, and we did not observe that the participant's knowledge from the previous study helped in any substantial way.

3.3 Tasks

Participants worked with jEdit, an open source text editor, which has also been used in previous lab [14] and version control [19] studies. Participants were provided an Eclipse workspace with the entire jEdit 4.3pre5 source, which is 54,720 non-comment, non-blank lines of Java.

To ensure that the tasks were the right length and difficulty and that they challenged developers in their ability to understand design, we iterated our tasks by piloting them with three pre-test subjects. After poor experiences with functional change tasks, we picked nonfunctional tasks focused on improving the design rather than implementing features or fixing bugs. We hoped this would challenge participants' ability to understand design more than fully specified changes to the application's behavior. Both tasks were designed to be architectural in nature by involving interactions between classes that we had identified as top level components on our component and connector diagrams. Many of the methods that participants studied were architecturally significant in participating in the connectors joining these components.

In both tasks, we provided design criticisms and corresponding code locations. Participants were instructed to "investigate why this is the case and implement a better design" and "make the design as ideal as possible by the criteria of performance, understandability, and reusuability". To ensure they knew that they

were expected to implement changes, they were instructed to "carefully budget your time to make your improved design as ideal as possible while carefully scoping your changes to what you can implement within your allotted time" while changing "as much or as little code as you'd like".

On the **FOLDS** task, participants investigated how fold level state was updated following edits to a file. jEdit allows hierarchical regions of text of the viewed file (e.g., a method body) to toggle between being "folded" up and hidden or viewed, by clicking on an arrow (or). Following an edit to a line, the line's fold level becomes invalid. When it is next requested by a call to `getFoldLevel`, it is recomputed and stored in a cache in `LineManager`, part of the buffer's implementation. If the fold level changes, a `fireFoldLevelChanged` event is sent. Subsequent calls to `getFoldLevel` retrieve the line's cached fold level from `LineManager` rather than recomputing it.

Participants were provided the following code excerpt:

```
/* force the fold levels to be updated. when painting
the last line of a buffer, Buffer.isFoldStart() doesn't
call getFoldLevel(), hence the foldLevelChanged() event
might not be sent for the previous line. */
buffer.getFoldLevel(delayedUpdateEnd);
```

This is a call in the `doDelayedUpdate` method from a class owned by JEditTextArea (responsible for editing) to the buffer (jEdit's term for a file). Participants were told that this call was "architecturally questionable" in changing "the buffer's state from a different component" and "clearly bad design" "using a getter method solely to change the state of the buffer and ignoring the information the getter method is supposed to be used to obtain".

Underlying the symptom of the problem (updating fold levels by calling a getter), the cause was the need for fold update to be triggered from this method. Participants were left to discover this and why it was bad. Folds are a responsibility of the buffer but the implementation has leaked into another component (`JEditTextArea`) because of this call's presence. Fold levels are lazily computed only when queried by `getFoldLevel`. The call is required due to this decision (it could be removed if fold levels were not lazily computed) and thus breaks information hiding [15]. `isFoldStart` does not call `getFoldLevel` when painting the last line of the buffer because it computes the fold level by comparing the current line's fold level with the next (undefined for the last line in a buffer) and instead always returns false. The call depends on this very private decision (it would not be necessary if `isFoldStart` were implemented differently) and this also breaks information hiding.

The **CARETS** task related to the status bar at the bottom of the jEdit window which displays the line and column of the caret (insertion point) and the scroll position of the window within the buffer. This is implemented, in part, using the `updateCaretStatus` method. Participants were asked to set a breakpoint on `updateCaretStatus`, make the buffer visible in jEdit, and observe that `updateCaretStatus` is called many times. Participants were instructed that this was bad from a performance perspective and "likely reflects deeper problems in the semantics of what the events that trigger these updates mean." The performance critique was contrived in that no extremely resource intensive operations were performed even though methods were needlessly executed. But an expert reported:

> But I've seen this situation before with something that was more directly expensive. – M interview

The **CARETS** task required understanding the design of the buffer switch process. Any action changing either the caret position or the scroll position must call `updateCaretStatus` to update the status bar. Buffer switches change both of these. They begin with a `setBuffer` call. Control then passes through nineteen methods on paths ultimately resulting in 6 or 7 `updateCaretStatus` calls. Many of these methods are also called for reasons other than buffer switches (including changes in text selection, window scrolling, or caret moves). Removing any calls to `updateCaretStatus` risks breaking these features.

We illustrate our results with think-aloud episodes which we label by participant, time within the task, and task (**C** for **CARETS**, **F** for **FOLDS**)(e.g., M 1:20(**C**) is expert participant "M" at time 1 hour, 20 minutes, doing the **C** = **CARETS** task).

3.4 Tools and Instrumentation

Participants were provided with the Eclipse 3.2.0 IDE and were allowed to use any Eclipse feature, take notes with Windows Notepad or on a piece of paper, and open files created by jEdit in Notepad or jEdit. To prevent searching for jEdit documentation, bugs, or other information that only some might think was relevant, participants were forbidden from using other applications, including web browsers. One participant asked and was allowed to see the JavaDoc for a collection class in a web browser. The experimenter answered questions about invoking specific Eclipse commands (e.g., how to stop the debugger or to use `System.err.println`() rather than `System.out.println`()) or what the task asked them to do, but not any other questions such as questions about the code (e.g., "is my understanding correct?") or strategies about how to use Eclipse to locate information (e.g., "how do I locate a method that triggered an event?").

Participants were recorded using a diverse set of recording devices so none of their actions would be lost. We used Camtasia to record the screen, a video camera of the participant's desk area to track referencing paper handouts and see which area of the screen was being viewed, and a second video camera to track information written on paper. Participants were asked to think aloud and prompted approximately every five minutes if they forgot to do so. Unfortunately, we prompted participants with "what are you trying to do?", leading some to talk more about their goals than the facts they had discovered. In retrospect, a better prompt might have been "what are you thinking about?"

3.5 Procedure

Participants first worked through a brief tutorial on Eclipse code navigation features (such as using the call hierarchy, navigating to method declarations, and reference searches) to ensure they effectively used Eclipse. To simulate some of the architectural knowledge that an experienced developer might possess, participants read a one page description of the responsibilities of eight important task relevant classes. Finally, they worked on a jEdit tutorial where they used the functionality they would be editing so that later testing would be easier. This portion of the study lasted approximately 30 minutes.

Next, participants received a sheet of paper describing the first task. Participants had as much time to read the task description as they liked. Participants then navigated to the code described in each of the tasks. On the **CARETS** task, they also tried out the

Table 2. Code changes implemented by participants, grouped by change and then sorted by years of industry experience, with total time on task. Changes in bold address the underlying design problem.

Participant(yrs industry exp)(time)	FOLDS task final code changes
a(0)(1:30)	All `getFoldLevel` callers check if fold update necessary and conditionally update
b(0)(1:11)	Update folds indirectly by firing the `foldLevelChanged` event
c(0.5)(1:18)	Renamed `getFoldLevel` to `updateGetFoldLevel`
e(2)(0:46)	Do not force fold update
f(2.5)(1:06)	Added debug statement, gave up
d(1.5)(0:44) g(2.5)(1:35) h(2.5)(1:34) i(3)(1:31) j(3)(0:53) K(3)(1:34)	Force fold update by calling method extracted from `getFoldLevel`
L(10)(1:35)	**Folds updated immediately after buffer changes by call from within `JEditBuffer`**
M(10.5)(1:14)	**Moved fold update to `isFoldStart` within `JEditBuffer`**

Participant(yrs industry exp)(time)	CARETS task final code changes
b(0)(1:34) c(0.5)(1:13) e(2)(1:18)	No changes
a(0)(1:30) d(1.5)(1:15) g(2.5)(1:33) h(2.5)(1:23)	Removed calls believed to be unnecessary.
f(2.5)(1:34)	Added class to log events that happened and detect if caret update should fire
i(3)(0:59) j(3)(1:03)	No changes, gave up
K(3)(1:35) L(10)(1:32) M(10.5)(1:35)	**Added field to stop caret updates during buffer switches**

behavior they were to change by setting a breakpoint and verifying that it was hit many times as the task description claimed. Participants were instructed that they had 1.5 hours to work but were given up to five extra minutes. Afterwards, participants were asked a series of exploratory interview questions about how they worked, what they found challenging, and ratings of how well they believed they did. Participants then received a clean Eclipse workspace and the description of the second task and began working on the second task. Materials are available online.[1]

We were successful at making our tasks challenging. While we expected some participants would be unable to make meaningful changes, we expected all participants to at least try to understand the code. However, one novice gave up on the **FOLDS** task, and two novices gave up on the **CARETS** task. They felt the code was too complicated for them to comprehend:

> It's too tough for me. I can't figure it out. There's bits and pieces that I understand but I don't understand precisely what the design issue is. – f 1:05(**F**)

An expert thought the **CARETS** task was realistic:

> That is just tough. Yikes, glad I'm not getting paid for this.– M 1:20(**C**)

> Yeah, this is realistic. I mean this is realistic on a bad day, at least in my assessment. – M interview(**C**)

Many participants were still working when time expired. Two **CARETS** participants (e, c) elected to describe in notes the list of changes they felt they did not have time to implement.

3.6 Analysis

Our analysis started with the low level data we recorded and built successively more abstract representations. We transcribed think-aloud recordings and screen capture video into 26 action logs consisting of a total of 11,821 lines. Every time a participant changed the method or field (referred to as a "member") visible in Eclipse, we added an entry naming the member and Eclipse command used to bring it into view. These included hitting breakpoints, stepping in the debugger, navigating using the call hierarchy or search results, going to declarations, navigating gutter references, and scrolling. We also coded edits, refactor commands, and running the program. We also noted goals participants appeared to be working towards.

Next, we used qualitative protocol analysis. We built a list of activities we saw developers engage in and coded what developers did using this model. Our analysis remained qualitative as we did not produce definitions sufficiently reliable to count and quantitatively compare activities. We discovered that many activities revolved around *facts* about the code. Participants chose methods to read, *seeking* facts they deemed relevant to the task. While reading methods, they sometimes *learned* facts which they believed with varying degrees of confidence. Participants felt some facts violated their design norms and wished to change them. Participants *explained* facts to understand how facts were related and the consequences of changing a fact. This sometimes generated hypotheses which led participants to seek evidence to confirm or reject facts. As participants learned more facts, they began to *propose* design changes that addressed their criticisms and task goals. Finally, participants *implemented* their proposals by editing code. When participants discovered facts leading them to believe their changes would not succeed, they removed the changes and proposed different changes.

Experts nearly constantly talked while most novices said nothing for minutes on end. This suggests that novices were more overwhelmed or spent more time immersed in details. When comparing experts and novices, we chose situations where some experts

[1] http://www.cs.cmu.edu/~tlatoza/fse07materials.html.

and some novices said something or situations where we could rely on what they did.

4. RESULTS

We first discuss the changes participants implemented. We then present the model we built to describe how our participants worked. We model developers as seeking facts, learning facts, critiquing facts, explaining facts, proposing facts, and implementing facts. We consider the structure of each of these activities in turn and differences between experts and novices.

4.1 Code Changes

Changes made by the experts addressed the cause of the underlying design problems. Changes made by the novices (if any) were inferior in that they only addressed the symptoms. We described the underlying design changes ignoring defects they may have introduced or whether they finished. We then clustered similar changes. Table 2 lists the final changes (if there were more than one) that the participants implemented or began implementing.

On the **FOLDS** task, one novice made no changes and gave up (f). Another (e) could not determine why the `getFoldLevel` call was necessary and removed it. The remaining novices changed the way in which `doDelayedUpdate` updated folds to address the symptom that a getter was being used purely to set. One (c) renamed the method to `updateGetFoldLevel` to indicate that it was not merely a getter. Another (b) literally interpreted the provided comment to mean that `doDelayedUpdate` needs to send the `fireFoldLevelChanged` event and created a method to do this. Six novices and one expert extracted an update method from `getFoldLevel` and had `doDelayedUpdate` update folds by calling this method. Changes made by two of the experts addressed the cause of the design problem by removing the need for `doDelayedUpdate` to force fold update. One (L) moved the fold update to two methods in `JEditBuffer` which are called after the buffer changes. Another (M) moved the fold update to `isFoldStart` within `JEditBuffer`. Both experts addressed the hidden design problems by removing the `getFoldLevel` call in `BufferHandler` that added questionable dependencies.

On the **CARETS** task, two novices (i, j) made no changes and gave up, and three made no changes but worked for the entire time (b, c, e). One (f) added a class to log `updateCaretStatus` calls with the (mistaken) intention to have it decide if `updateCaretStatus` should proceed from other recent calls. Four novices removed calls they believed were redundant. Expert changes differed from novice changes in starting and stopping caret updates using a field. This approach alone addressed the cause of the design problem in that it could reduce the number of calls to one.

4.2 Seeking Facts

Participants began their tasks navigating from the methods we provided to methods and fields they believed likely to reveal relevant facts about the code. Participants visited between 5 and 59 members on the **FOLDS** task and 25 and 41 members on the **CARETS** task (Figure 1). There was no effect of experience on how many members participants visited, and participants visited similar numbers on both tasks. Thus, experts' superior changes were not due to reading more members but from selecting better members to read and learning more from reading them.

Participants made *path choice decisions* when choosing between locations in which to seek, deciding if seeking was likely to discover a useful fact, or choosing between seeking and implementing the current change. A novice abandoned seeking in a location:

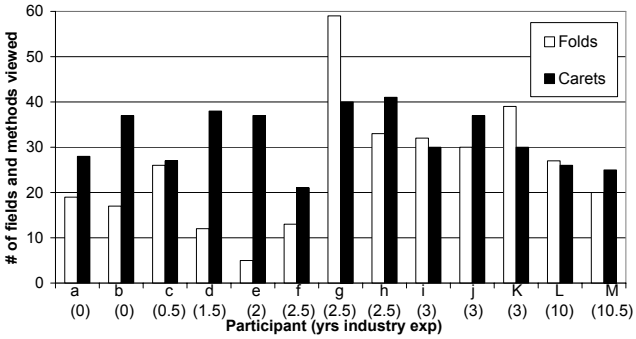

Figure 1. Total distinct fields and methods viewed by each participant sorted by years of industry experience.

> So after it runs runnable thread, I get three extra calls to the update caret method. I don't know what thread it is --. I can go in and find out more, but I don't think it is the unnecessary type that I'm looking for. – d 1:09(**C**)

An expert considered whether a change should be implemented or whether better alternatives should first be sought:

> I can reduce the event firing from here, huh, is that even the right path to go down? Let's see, we've got setCaretPosition, no, oh wait what about, ohh setCaretPosition is the one that is called by many people. Ok, I'm going to give it a long hard look at the finishCaretEvent, no finishCaretUpdate. – K 0:41(**C**)

We investigated members visited only by novices to understand why novices wasted time visiting members that experts did not need to visit. On the **CARETS** task, there were 12 members visited by no experts which were visited by at least three of the ten novices. One was a class definition, which novices visited more because they used the open class Eclipse command more. 9 were transitive callers of `updateCaretStatus` which novices navigated to more because of inferior navigation strategies. The remaining two members were the most interesting. One was a field – `showCaretStatus` – which guarded the body of `updateCaretStatus`. The other was `propertiesChanged` where `showCaretStatus` was initialized at startup. One novice (f) stumbled into `propertiesChanged` and quickly left it. Four other participants read one or both of these methods because they were interested in the meaning of `showCaretStatus`. One (g) spent a minute looking for `showCaretStatus` references. Another (c) spent two minutes looking at how `propertiesChanged` worked. Two (a, h) spent 7 minutes understanding in detail how `propertiesChanged` worked:

> So I guess the whole debug that is remaining is that when I switch buffers this showCaretStatus variable needs to be reset as soon as I update the caret position. – h 0:58(**C**)

Novices seemed to perceive `showCaretStatus` as indicating the presence of a changeable fact that might help them reduce `updateCaretStatus` calls. Reading `propertiesChanged`, they eventually discovered `showCaretStatus` merely controls whether caret and scroll position is displayed on the status bar and would not be helpful for their task. That experts never wasted time reading these members suggests that knowledge helped them guess from the field's identifier and use that showCaretStatus did not turn on and off updates during an event but rather in general. This suggests that knowledge helps experts predict what code

does before reading it, thereby preventing wasted time reading irrelevant methods.

4.3 Learning Facts

When reading methods, participants found interesting facts that confirmed or disconfirmed expectations:

> These all look like mutators on the buffer. So that makes sense. So at the end of the mutating operation on buffer, it's going to end in doDelayedUpdate. – L 0:06(**F**)

Facts played a variety of roles. Facts were *changeable* when developers believed alternatives to them might help accomplish their task goals. Others acted as *constraints* which suggested that some changes would break them and should not be chosen. Others made changes *expensive* by suggesting lots of investigation would be required:

> Wow, many, many, many methods call getFoldLevel and that is not good because it's going to be hard to figure out what all of those are. – K 0:02(**F**)

Facts also differed in the degree of certainty with which participants believed them. Some were *hypotheses* thought likely to be true. Some hypotheses were generated from knowledge about how the application would probably have been built to satisfy its requirements:

> So mouse released represents the bottom of the tree for certain. setSelectedIndex is part of JComboBox.fireAction event. It's possible that we're getting multiple action handlers involved here, but let's assume that that is not the case. – M 0:16(**C**)

Other facts were directly *observed* in code. But many relied on both observation and knowledge-driven speculation. Figure 2 lists some facts found by an expert.

Experts more frequently and rapidly used facts at higher levels of abstraction which focused on the important and relevant parts of code rather than irrelevant implementation details. For example, experts and novices described `getFoldLevel` very differently. One minute into the task, an expert described `getFoldLevel`:

> Well this is just updating a cache. So, what we're upset about is that you want to issue an event and you are doing it by forcing an update of the cache for the fold level of a particular line. – M 0:01(**F**)

After 38 minutes in the task and 10 minutes reading `getFoldLevel`, a novice still had not figured out how it changed state:

> What it did was it compute I mean computes the new line number and fires an event. But I didn't see it change any state. – b 0:38 (**F**)

51 minutes into the task, after over 12 minutes staring at `getFoldLevel`, and having read numerous callers and callees, a different novice was still stuck at the statement level, never describing it as caching:

> So what it does, it starts off from this line, it has this firstInvalidFoldLevel, it goes through all these lines, it checks whether this fold information is correct or not, which is this newFoldLevel, this is supposed to be the correct fold level. If that is not the case in the data structure, it needs to change the state of the buffer. It creates this, it does this change, it sets the fold level of that line to the new fold level. – h 0:51(**F**)

These differences suggest that schemas, such as caching, allow experts to see design abstractions and chunk individual statements using these schemas. Applying the caching schema helped the expert infer the intent of the code. Lacking the expert's schema, novices were not able to uncover this intent and painfully worked through the code statement by statement**.**

1. HACK: `getFoldLevel` has effects
2. Buffer mutating operations result in a `doDelayedUpdate` call
3. HYP: `doDelayedUpdate` does changes that happen later
4. Many methods call `getFoldLevel`
5. Folds invalidated by buffer changes are updated on screen. EXPLAINS 2, 1, 8
7. `getFoldLevel` updates a fold data structure EXPLAINS 1
8. `getFoldLevel` fires events
10. CRIT: `getFoldLevel` determines if folds must be set
11. CRIT: `doDelayedUpdate` triggers fold update
12. `isFoldStart` calls `getFoldLevel` on startup
13. `getFoldLevel` mutually recursive with `FoldHandler.getFoldLevel`
14. Folds are initialized at startup EXPLAINS 12
15. `BufferHandler` is only buffer listener
16. Either `fireContentInserted` or `fireContentRemoved` is called after every buffer mutating operation

Figure 2. Some facts found by expert participant L in the first 41 minutes of the FOLDS task in the order they were discovered. Facts are labeled with hack, hypothesis, and critique roles and the explanation of the relationships.

4.4 Critiquing Facts

Consistent with instructions to improve the design, participants used their good design norms to criticize facts. Growing skeptical of design choices they perceived the original authors had made, they designated those as *hacks:*

> And this guy who is probably hacking away… This started out with this thing as just a getter and said, oh look when you're getting the fold level there can be a case where your data is now invalid so I might as well go fix it up right here. And he might have wandered himself into the bad design situation that we've got right now. – L 0:16(**F**)

In their criticisms, participants exhibited design knowledge by perceiving a design choice, alternatives, and justifying the inferiority of the current choice. A single expert perceived this design choice:

> And the second thing that I don't like is that it is firing these updates. It seems like when you're making the edit, that in order to keep the responsibilities of these guys very simple, when you're making the edit, the people that care about that would be notified. – L 0:26(**F**)

Several novice criticisms resulted from missing design knowledge:

> It just seems really confusing for me to have this exact same method with the exact same parameters, they both have the handleMessage. I should investigate that. Hold on. – c 0:40(**C**)

Only after investigation did the novice realize these methods implemented the same interface.

4.5 Explaining Facts

Participants explained the rationale of facts they learned:

> So because this is lazily evaluated, which you probably want to do for performance reasons anyway, you're always going to have the risk that a get is going to fire an event in any case. – M 0:09(**F**)

Explanations established traceability from low level facts about the implementation towards motivating requirements. These dependencies became important when a participant wished to change a fact. Because of these dependencies, changing a fact risks changing other facts. Explanations generate these facts.

Participants also applied explanations top down to hypothesize how the code was likely built to satisfy higher level constraints:

> He must be either firing events to tell people to update. Or somehow there must be some other code to then update the display. But it looks like the event firing is happening inside there. – L 0:17(**F**)

When participants believed *false* facts, explanations produced more false facts which were then critiqued or used as constraints. These formed breakdown chains [10] where the participants' model of the code had gone badly awry. A developer explained a false callgraph fact as due to something triggering a buffer edit:

> 'Cause I'm thinking that when I perform the action of switching from one buffer to another buffer, somewhere it calls a method that indicates that the buffer has been edited. But I didn't edit the buffer. I'm just switching between buffers. So that has to be removed. – d 0:30(**C**)

This hypothesized call from a buffer edit did not exist because the call he used it to explain did not exist.

Participants reasoned using *code* facts which described the implementation and *requirements* facts which described application behavior in terms of the domain. False requirements led to missed constraints. A novice forgot that the instructions stated that the status bar displays caret and scroll position:

> This is, I think this is completely unnecessary because why would a scrolling event cause a caret update. Like if I'm just scrolling, by ---, it doesn't change the caret offset. So I think I should just get rid of this one actually. – d 0:33(**C**)

Participants with a changeable fact that they could not explain faced a choice – optimistically assume it was *overlooked* by the original developer or pessimistically assume it was *intended* to satisfy a hidden constraint. Overlooked facts are true because they happen to be true – changing them does not affect other facts. Intended facts can be safely changed only when the developer is able to generate an alternative fact that still satisfies all the constraints. Optimistic assumptions caused bugs. Pessimistic assumptions led developers to abandon considered changes, *freezing* the fact and preventing consideration of changes:

> So here they're basically deselecting everything and then they're going to reselect everything. So initially I'm going to ignore that because maybe that's intentional by the designer because maybe they would want to if there's an error switching or --- reading from file. – a 1:25(**C**)

Participants investigated hypothesized constraints before concluding none existed and the fact was overlooked.

Some participants used beliefs about the abilities of the original developers to help distinguish intended and overlooked facts. An expert attempted to understand why an original developer had chosen a less desirable decision over an obvious decision:

> Why wouldn't they call it? Now, can I test this? So why if you know the answer to the problem, do you put the code in the wrong place and then leave a comment? That's not like these people. – M 0:35(**F**)

The expert believed the decision could not possibly be overlooked but must be intended, suggesting the search for a hidden constraint must continue. Subsequent discovery of a second example where the original developer overlooked an obviously better decision revised his beliefs:

> What a horrible little thing to do. Ok, that changes my view on the coding style. – M 0:37(**F**)

Participants gambled when deciding if a proposed change would work based on information they did not yet have. Explanations helped predict the probability a change would succeed. One expert implementing a change found it unexpectedly difficult. He became concerned that a fact that he believed to be overlooked was intended and that his work implementing the change would be wasted when they discovered a frozen constraint which had prevented the original developer from making the same change. He presciently predicted, for the wrong reasons, that his 23 minutes implementing the change would be wasted:

> [laughing] This is never going to work, the thing is there's just all this mess going on with this caret listening… If it was just as easy as getting EditBus messages and updating the caret it would be straightforward. And the other question I've got, is that there's already CaretListener. And why doesn't it just… do caret listening itself? – L 0:41(**C**)

When developers had a hypothesized explanation of the underlying cause, they rejected changes that did not address this cause, even lacking evidence supporting their hypothesis:

> Somehow if I can track it down from the origin of when the event occurs and from there I can pass in a Boolean false to every function call except for one. So it's trickle down... But that seems like a hack because this is called 4 times and it shouldn't be. – j 0:54(**C**)

After proposing several similar changes, he gave up lacking a strategy to check his hypothesis.

In understanding why two experts made different changes than the other participants on the **FOLDS** task, we observed that both better understood why the call was necessary and sought a better way for this constraint to be satisfied subject to their critiques. One expert explained the call using a model of how the application behaved:

> What's going on is that when you're inserting text you could actually be doing something that makes the folds status wrong. So, if in our example here, in the quick brown fox. If fox is under brown and I'm right at fox and I hit backspace. Then I would need to update my fold display to reflect the new reality, which is that it's in a different place. – L 0:15(**F**)

No other participant produced this explanation. The expert subsequently mapped specific code locations to serving specific goals and constantly talked about how each of the locations served a purpose in satisfying this requirement. This unique explanation of why the call was necessary allowed him to propose a unique solution – moving fold update from its current, poorly chosen location to a point earlier in the process.

In response to the task description's vague instructions that the `getFoldLevel` call was "architecturally questionable", another expert asked a unique question about `BufferHandler`:

> So what I need to do is figure out how it's using its buffer. Is this the only mutation that they're doing? – M 0:18(**F**)

This generated a unique critique – the `getFoldLevel` call caused `BufferHandler` to retain an "architecturally weird" buffer reference. He then moved fold updating to `isFoldStart` to address this critique.

Three novices who were tantalizingly close to these changes abandoned them following pessimistic assumptions. One novice (g) implemented moving fold update but gave up when he believed a bug indicated he was breaking a frozen hidden constraint.

Another (j) tried to explain why the call was in `BufferHandler` by understanding how its parameter was computed until he abandoned this path. Another (h) failed to explain the purpose of `BufferHandler` and felt this hidden constraint made a change too risky. An expert (K) abandoned considering this change when he stated the false hypothesis, without checking it, that `BufferHandler` was intended to change the fold level, rejecting the task's architectural criticism.

4.6 Proposing Facts

Participants proposed design changes, composed of individual fact changes, to accomplish their task goals and address problems they had perceived. Participants usually first talked about a summary of what they had learned and then proposed a change. Changes often began as vague goals, generating hypotheses, and were then refined by learned facts. One expert (L) proposed six changes in 20 minutes before discovering one they believed they had time to implement. Many changes reflected the application of design patterns [6] they had seen before:

> When I do this, I have two different styles; I have two different methods. So there might be something that directly manipulates a variable and then there's like a publicly visible, sorry if somebody calls like setX, I update, send notifications that x has changed or whatever, but if I'm doing something internally I munge, munge, munge and then manually tell people at the end. – L 1:19(**C**)

Novice proposals often did not solve the problem and worked out implementation details rather than considering general patterns:

> How about maintaining for every View, for every buffer, maintaining the caret position in a hashtable. ... The key would be the buffer object and the value would be, say I have the x,y positions of the caret. That's all. I'll have one hashtable, a static hashtable for the application. – e 1:14(**C**).

Of the 29 proposed changes on the carets task, experts (K, L, M) were the only participants to propose using a field to start and stop caret updates. Other proposals included removing redundant calls, passing a Boolean of whether to call `updateCaretStatus` to all of its callers, and recording caret information.

4.7 Implementing Facts

We observed one situation where many participants made the same change – 8 participants extracted a fold update method from `getFoldLevel`. Table 3 shows that more experienced participants did this more quickly. An expert extracted it merely to better understand it. Other participants intended it as their final change. Participants taking longer appeared aimless or confused, spent tens of seconds staring at code, revisited perceived decisions, visited callees, and moved statements between methods. Participants taking less time recognized the block of code they wanted to extract and used the Eclipse command "Extract Method". This is consistent with a *chunking* interpretation – experts encoded what the code did using more abstract facts. Novices saw the code statement by statement and the interrelationships between statements and got bogged down considering changes at this level.

Table 3. Minutes to extract update method from getFoldLevel for participants(yrs industry experience) who tried to do this, sorted by experience with experts in bold

a(0)	d(1.5)	g(2.5)	h(2.5)	j(3)	i(3)	**K(3)**	**L(10)**
10	13	4	11	9	4	3	4

5. EXTERNAL VALIDITY

By studying developers in a lab, rather than in the field, participants worked differently in ways which likely made the tasks more challenging. Participants were new to the application and code and could not rely on anything more than the rudimentary information we provided about the design, architecture, and features of jEdit to reason about the application. Participants were asked to make changes designed to require substantial understanding of the design. Developers might typically have much more experience before taking on such changes. Otherwise, such tasks are often used to learn the code with much more relaxed time requirements than our hour and a half tasks. Developers may also answer tough questions by seeking out other developers who may know the code better and provide important insights [12]. Developers working on code with unit tests might learn why functionality is necessary by commenting it out and finding failing unit tests. By asking participants to make the design as ideal as possible, we may have caused the participants to spend more time or be more careful with the design implications of their changes than they would have otherwise been. But, as our aim was to model the program comprehension process and expertise effects, rather than measure the magnitude of these effects, we do not believe these concerns call into question any of our findings.

6. DISCUSSION

We discovered that program comprehension is driven by beliefs about *facts*. Dependencies between decisions took the form of explanations that developers used to form chains of facts and elicit constraints they would need to respect in their proposals and changes. A key driver of the program comprehension process was uncertainty. Developers chose how much confidence to express in their hypotheses and made path choice decisions about whether to seek evidence to support them. Developers were uncertain whether a hidden constraint would force them to abandon their changes or unknowingly break a requirement. Developers used sophisticated strategies to judge the likelihood of a hidden constraint's presence such as judging the skill level of the original developer.

An interesting finding is that many of the facts developers thought about took the form of simple predicates about the code. The simplicity of these facts suggests simple analyses could help discover and visualize them. When understanding a method, an expert thought about facts about it (e.g., `getFoldLevel` has effects), explanation relationships with other facts (e.g. `doDelayedUpdate` calls `getFoldLevel` to update folds), critiques (e.g., `getFoldLevel` should not have effects), and design changes resolving these critiques subject to constraints. A tool that helps query for some of these facts might reduce the amount of time developers spent reading code or increase the number of hypotheses that they check. A tool externalizing these facts might help make it easier for developers to remember them and return to the code associated with them. A previous study [11] viewed developers' "working set" of task-relevant facts as regions of code and proposed an editor to externalize these. We found developers abstractly discussing sets of statements with facts, perhaps because our design tasks focused on constraints while the previous study [11] focused on changeable facts. When developers focus on facts, not statements, externalized views could be more compact by showing only relevant facts. Design rationale systems have long sought to capture explanations of facts but

were designed for up-front design and design meetings and work only with requirements and high level design and requirements facts [13]. A tool that captured explanation linkages might make it easier to find these later.

When developers considered alternatives, facts played the role of design decisions. This suggests a measurable definition of information hiding – a fact is hidden during a task when a developer does not think about it. This differs from defining information hiding in terms of methods read by a developer. A developer may hypothesize constraints without ever reading or even locating the code embodying these constraints, but these facts may still profoundly influence design choices. Conversely, a developer may read a method at a high level of abstraction and not notice or consider detailed facts that are explained by the facts of interest.

While our observations have allowed us to theorize, our results do not carry the certainty of controlled experiments. All of the differences we observed were interrelated in complex ways and could have causes that we were not able to discern. While we have identified interesting, novel differences and built a theory to explain them, controlled experiments would be desirable to test these differences.

7. CONCLUSIONS

Drawing on design decisions and dependency from normative models of how developers ought to design, we found these helped better describe how developers actually worked. Our results suggest that tools that allow developers to directly work with relevant facts could help them work more effectively.

8. ACKNOWLEDGEMENTS

We thank the participants in our study for their participation. We thank Andrew Ko and Vibha Sazawal for helpful ideas and discussion. We thank Andrew Ko, Chris Scaffidi, Ciera Jaspan, and Marwan Abi-Antoun for helpful comments on earlier drafts of this paper. This research was funded in part by NSF grants IIS-0329090 and IIS-0534656 and by the EUSES consortium under NSF ITR CCR-0324770. The first author was also supported by a NSF Fellowship. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

9. REFERENCES

[1] L. M. Berlin. Beyond Program Understanding: A Look at Programming Expertise in Industry. *Empirical Studies of Programmers: Fifth Workshop*, 6-25, 1993.

[2] P. Clements, F. Bachman, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford.*Documenting Software Architectures: Views and Beyond*.Addison-Wesley, 2002.

[3] W. G. Chase and H. A. Simon. Perception in chess. In *Cognitive Psychology*, 1973.

[4] S. P. Davies. Externalizing information during coding activities: Effects of expertise, environment and task. In *Empirical Studies of Programmers: Fifth Workshop*, 42–61, 1993.

[5] F. Détienne. Program understanding and knowledge organization: the influence of acquired schemata. In *Cognitive Ergonomics: Understanding, Learning and Designing Human-Computer Interaction,* 245-256, 1990.

[6] F. Détienne. *Software Design---Cognitive Aspects*. Springer-Verlag New York, Inc, 2002.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reusuable Object-Oriented Software*. Addison-Wesley, 1995.

[7] L. Gugerty and G. M. Olson. Comprehension differences in debugging by skilled and novice programmers. In *Empirical Studies of Programmers,* 13-27, 1986.

[8] B. A. Kitchenham, T. Dybå, and M. Jørgensen. Evidence-Based Software Engineering. In *Proc. Int'l Conf. Software Eng (ICSE),* 273-281, 2004.

[9] A. J. Ko., R. DeLine, and G. Venolia. Information Needs in Collocated Software Development Teams. In *Proc. Int'l Conf. Software Eng (ICSE)*, May 20-26, 2007.

[10] A. J. Ko and B. A. Myers. A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems. In *Journal of Visual Languages and Computing*, 16, 1-2, 41-84, 2005.

[11] A. J. Ko, B. A. Myers, M. Coblenz, and H. H. Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. In *IEEE Transactions on Software Engineering (TSE),* 32(12), 971-987, 2006.

[12] T. LaToza, G. Venolia, and R. DeLine. Maintaining Mental Models: A Study of Developer Work Habits. In *Proc. Int'l Conf. Software Eng (ICSE)*., 492-501, 2006.

[13] T. P. Moran and J. M. Carroll, Eds. *Design Rationale: Concepts, Techniques, and Use*. Lawrence Erlbaum Associates, Inc, 1996.

[14] M. P. Robillard, W. Coelho, G. C. Murphy. How Effective Developers Investigate Source Code: An Exploratory Study. In *IEEE Transactions on Software Engineering (TSE)*, vol. 30, no. 12, 889-903, Dec. 2004.

[15] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. In *Communications of the ACM*, Vol. 15, No. 12, 1053 – 1058, December 1972.

[16] D. E. Perry. and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes* 17, 4, 40-52, Oct. 1992.

[17] K.J. Sullivan, W.G. Griswold, Y. Cai and B. Hallen. The Structure and Value of Modularity in Software Design. In *Foundations of Software Engineering (ESEC/FSE),* September 2001.

[18] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc, 1996.

[19] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. In *IEEE Transactions on Software Engineering (TSE),* 31, 429-445, June 2005.

[20] A.M. Vans, A. von Mayrhauser, and G. Somlo. Program Understanding Behavior during Corrective Maintenance of Large-Scale Software. In *Int'l J. Human-Computer Studies*, vol. 51, no. 1, 31-70, July 1999.